# Vérification formelle d'extractions de racines

Yves Bertot

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Vérification formelle d'extractions de racines*

Yves Bertot

**N° 5344**

Octobre 2004

Thème SYM


ISSN 0249-6399 ISRN INRIA/RR--5344--FR

# Vérification formelle d'extractions de racines

Yves Bertot

Thème SYM — Systèmes symboliques
Projet Lemme

Rapport de recherche n° 5344 — Octobre 2004 — 23 pages

**Résumé :** Nous décrivons la vérification formelle d'algorithmes de calcul de racines carrées, cubiques, et énièmes dans un cadre fonctionnel. Nous montrons que les premiers algorithmes se décrivent bien en utilisant la représentation binaire des entiers, qui permet en outre d'assurer la terminaison de ces algorithmes. La même structure est sous-jacente à l'algorithme de racine énième, mais la terminaison de l'algorithme est vérifiée en utilisant des outils plus complexes. Le travail de vérification formelle a été effectué en utilisant le système Coq.

**Mots-clés :** Vérification formelles de logiciel, Calcul des constructions, arithmétique des ordinateurs

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

# Formal verification of root computation

**Abstract:** This paper describes the formal verification of algorithms to compute square roots, cubic roots and nth root in a functional framework. We show that the first algorithms are easily described by using the binary representation of integers, a solution that moreover makes it possible to ensure the termination of these algorithmes. The same structure underlies the algorith for nth roots, but ensuring that the algorithm terminates requires more complex tools. The formal verification work was done using the Coq system.

**Key-words:** Formal software verification, Calculus of constructions, Computer arithmetics

# 1 Introduction

Parce qu'elle est très proche de la structure des polynômes, la représentation des nombre par position dans une base donnée est bien adaptée pour un grand nombre d'opérations, comme la multiplication, l'addition ou la division. Parmi les opération fréquemment utiisées dans le calcul scientifique, l'extraction de racines, en particuler l'extraction de racines carrées, n'échappe pas à la règle.

Nous avons déjà étudié les algorithmes d'extraction de racines carrées pour les grands en nombres en base arbitraire dans un travail précédent [5]. Cette étude poussait le travail jusqu'à la description d'un algorithme impératif et incluait en particulier la vérification formelle d'une optimisation fine dans l'utilisation de l'espace mémoire. Dans ce travail précédent, l'approche est de type *diviser pour régner* au sens où la moitié des chiffres du résultat est déterminée en une itération (sachant que l'autre moitié est traitée par un appel récursif). Le nombre d'appels récursifs de l'algorithme est proportionnel au logarithme du nombre de chiffres utilisés pour représenter la donnée initiale.

Le travail présenté dans ici reprend l'algorithme d'extraction de racine carrée dans une forme plus basique, où un seul bit est déterminé à chaque itération. Il est assez facile d'étendre cet algorithme pour obtenir un algorithme d'extraction de racines cubiques, puis un algorithme d'extraction de racines énièmes qui peut éventuellement fonctionner pour de très grand nombres, avec un nombre d'appels récursifs cette fois-ci proportionnel au nombre de chiffres utilisés pour représenter la donnée initiale.

Du point de vue de la vérification formelle en théorie des types, les algorithmes de racine carrée et cubique et l'algorithme de racine énième ne sont pas dans la même catégorie. Les deux premiers se décrivent comme des algorithmes récursifs *structurels* pour lesquels le raisonnement est aisé, car il suffit d'effectuer des raisonnements par récurrence sur la structure des données. De tels raisonnements sont particulièrement bien supportés dans les systèmes de démonstration assistée par ordinateur en général. Une légère difficulté apparait dans la mesure où les appels récursifs ne se font pas en suivant directement la structure des données, mais nous montrons comment reprendre une technique déjà présentée dans [4] pour obtenir les outils de preuve adaptés. En suivant cette technique, nous arrivons à démontrer la correction de nos algorithmes de racine carrée et de racine cubique en suivant exactement le même plan de preuve.

Pour la racine énième, le problème se pose différemment, car l'argument de l'appel récursif est obtenu par l'intermédiaire d'une fonction de division. La bonne définition de l'algorithme, et en particulier l'assurance qu'il termine toujours, ne peut plus être reconnue par simple observation de la structure, mais doit reposer sur des propriétés de décroissance dérivées des propriétés de la fonction de division. Le problème est plus complexe car la définition de l'algorithme comme une fonction récursive doit alors combiner le contenu calculatoire de l'algorithme et la preuve que l'algorithme termine. Cette nécessité de combiner des concepts habituellement considérés séparément rend souvent la description de ce type de fonction récursive difficile. Dans cet article, nous utilisons deux outils que nous avons mis au point dans des travaux précédents et dont l'objectif est de réduire le niveau de compétence requis du programmeur pour la description des fonctions récursives et la preuve de leur propriétés.

Nous ne décrivons qu'une partie de la formalisation complète, en précisant surtout la façon dont les différentes fonctions sont décrites et les théorèmes qui ont été démontrés. La structure des démonstrations est seulement esquissée, mais les démonstrations sont disponibles sur internet, de sorte que cet article peut aussi être utilisé comme exemple didactique pour la description de fonctions récursives générales dans le calcul des constructions inductives.

Les fichiers de démonstration formelle sont disponibles sur internet à l'adresse suivante : `ftp ://ftp-sop.inria.r/lemme/Yves.Bertot/roots.tar.gz`

# 2 racine carrée

## 2.1 description informelle de l'algorithme

L'algorithme que nous étudions fait partie du folklore enseigné dans le secondaire il y a encore quelques dizaines d'années. Bien sûr, notre culture nous amène à travailler en base 10, ce qui constitue une différence notable avec l'algorithme que nous décrirons ici, mais le principe reste le même. La première étape consiste à grouper les chiffres du nombre considéré deux par deux, en commençant par les chiffres de la droite (donc en commençant par les chiffres de poids faibles). Il faut ensuite calculer la racine carrée du nombre composé du groupe le plus à gauche, qui est constitué de un ou deux chiffres, en conservant d'une part la partie entière de cette racine carrée et d'autre part le reste. On procède ensuite récursivement de la façon suivante : On accole au reste (du coté des chiffres des poids faibles) un groupe de deux chiffres venant de la donnée initiale, on multiplie par deux la valeur courante de la racine carrée, et l'on cherche le chiffre le plus grand tel que ce chiffre accolé au double de la racine carrée, le tout multiplié par le même chiffre soit plus petit que le reste. La nouvelle racine carrée est obtenu en accolant ce chiffre à l'ancienne racine carrée. Le nouveau reste est la différence obtenue au moment de la comparaison, et l'on peut reprendre l'algorithme récursivement.

Voici un exemple. Nous voulons calculer la racine carrée du nombre 45678. Le groupement des chiffres produit les groupes 78, 56, et 4. Nous commençons par calculer la racine carrée de 4 (c'est 2 et le reste est 0). Nous amenons ensuite le groupe de deux chiffres 56 à coté de 0 (nous obtenons ainsi le nombre 56), et par ailleurs nous calculons le double de 2, c'est 4, et nous cherchons le chiffre $c$ tel que $4c \times c$ soit inférieur à 56. Nous trouvons le chiffre 1 (car $41 \leq 56 < 42 * 2$). La nouvelle racine carrée est 21, le nouveau reste est 15. A la deuxième étape récursive, nous accolons 78 à 15 pour obtenir le nombre 1578, et nous cherchons le chiffre $c$ le plus grand tel que $42c \times c$ soit inférieur à 1578. Nous trouvons le chiffre 3 et le reste est 309. Le lecteur suspicieux pourra vérifier que $21^2 + 15 = 456$ et $213^2 + 309 = 45678$.

Si l'on cherche à décomposer cet algorithme de façon récursive, il est judicieux de le faire de façon abstraite vis-à-vis de la base utilisée pour écrire le nombre Notons $\beta$ cette base. La première étape de groupage des chiffres deux par deux correspond en fait à des divisions successives par le carré de la base. À chaque étape récursive, on considère donc que le nombre étudié $N$ est supérieur à $\beta^2$, que l'on a décomposé ce nombre en deux fragments $N'$ et $N''$

tels que $N = N'\beta^2 + N''$ et que $N''$ est inférieur à $\beta^2$ et que l'on a déjà calculé la racine carrée de $N'$ pour obtenir deux nombres $s'$ et $r'$ tels que $N'' = s'^2 + r'$ et $(s'+1)^2 > N''$. L'opération d'accolement d'un groupe de deux chiffres correspond à la fabrication du nombre $r' \times \beta^2 + N''$, le chiffre cherché $c$ est le plus grand tel que $(2 \times s' \times \beta + c) * c$ est inférieur à $r' \times \beta^2 + N''$ et on obtient un nouveau reste $r'' = (r' \times \beta^2 + N'') - (2 \times s' \times \beta + c) * c$. La combinaison de ces équations donne bien $N = (s' \times \beta + c)^2 + r''$.

Evidemment, cet algorithme fonctionne de façon similaire dans n'importe quelle base, mais en base 2 les chiffres possibles pour $c$ sont plus réduits et il n'est pas nécessaire de faire de multiplication supplémentaire. il suffit de vérifier si $4 * s' + 1$ est plus grand ou plus petit que le nombre obtenu en accolant les deux derniers bits au reste de l'opération précédente.

## 2.2 Description en théorie des types

Notre description formelle a été effectuée dans le calcul des constructions inductives à l'aide du système Coq [8]. Cette théorie permet de décrire des algorithmes fonctionnels et d'en démontrer les propriétés. Les calculs récursifs sont autorisés dans cette théorie, mais des contraintes sont imposées pour assurer que les calculs terminent. Ces contraintes sur la notion de type inductif. Intuitivement, chaque élément d'un type inductif est représentable comme un arbre et les les fonctions récursives ne sont autorisées à effectuer des appels récursifs que sur des sous-arbres de l'argument initial. De même que pour les langages de programmation fonctionnels typés de la famille de ML et Haskell [19, 7, 18], les analyses de données peuvent être effectuées par filtrage, ce qui permet d'écrire une fonction en donnant les différents cas de son comportement.

Le type inductif `positive` est utilisé pour représenter les nombres entiers positifs non-nuls codés de façon binaire. Ce type inductif repose sur la constatation que tout nombre entier positif non-nul est soit le nombre 1, soit le double d'un nombre entier positif non-nul, soit le double d'un nombre entier positif non-nul plus 1. Les deux derniers cas montrent bien que le type est récursif, puisqu'il faut déjà disposer d'un nombre positif pour en obtenir le double ou le double plus 1.

Dans le système Coq chacun de ces cas et décrit comme un constructeur du type `positive`, le cas 1 est appelé `xH`, le cas "double" est appelé `xO` et le cas "double plus 1" est appelé `xI`. Le système Coq fournit également un type `Z` pour représenter tous les types entiers, négatifs, nuls ou positif et la fonction `Zpos` (qui est un constructeur de `Z`) établit l'injection naturelle des entiers de type `positive` dans le type `Z`.

Lorsqu'un nombre est plus grand que 4, il y quatre possibilités différentes pour les deux bits de poids faible. Le nombre $4 \times p + 2$ sera alors écrit `xO(xI p)`, par exemple. Du point de vue de la récursion, `p` apparait bien comme un sous-terme de `xO(xI p)`. Pour rendre explicite le fait que les appels récursifs sont bien effectués sur des sous-termes, nous sommes obligés de répéter quatre instances du code effectué dans les appels récursifs, un pour chaque valeur possible des deux bits de poids faible.

```
Definition mod4 (x:positive) : Z :=
  match x with
```

```
    1%positive => 1 | 2%positive => 2 | 3%positive => 3
    | x0(x0 p) => 0%Z
    | x0(xI p) => 2%Z
    | xI(x0 p) => 1%Z
    | xI(xI p) => 3%Z
  end.

Definition sqrt_aux (s' r' n'':Z) : Z*Z :=
  if Z_le_gt_dec (4*s'+1) (4*r'+n'') then
     (2*s'+1, (4*r'+n'')-(4*s'+1))%Z
  else
     (2*s', 4*r'+n'')%Z.

Fixpoint sqrt (x:positive) : Z*Z :=
  match x with
    1 => (1,0)%Z
  | 2 => (1,1)%Z
  | 3 => (1,2)%Z
  | x0(x0 p) => let (s',r') := sqrt p in sqrt_aux s' r' (mod4 x)
  | x0(xI p) => let (s',r') := sqrt p in sqrt_aux s' r' (mod4 x)
  | xI(x0 p) => let (s',r') := sqrt p in sqrt_aux s' r' (mod4 x)
  | xI(xI p) => let (s',r') := sqrt p in sqrt_aux s' r' (mod4 x)
  end.
```

Nous utilisons la fonction `Z_le_gt_dec` fournie dans les bibliothèques de Coq et qui retourne une valeur positive lorsque son premier argument est inférieur au second.

On peut vérifier par quelques tests que les valeurs retournées par `sqrt` sont satisfaisante, mais nous voulons maintenant obtenir une preuve formelle qui assure ce résultat pour toutes les valeurs possibles de l'argument initial.

## 2.3 Techniques de démonstration

Pour chaque définition de type inductif le système Coq fournit un principe de récurrence adapté pour le raisonnement sur ce type. Intuitivement, ce principe de récurrence indique que toute propriété qui est conservée par les constructeurs du type est vraie pour tous les éléments du types. Pour chaque constructeur du type inductif qui présente de la récursion, ce principe de récurrence permet de raisonner avec l'hypothèse que la propriété est déjà vérifiée pour les sous-terme, cette hypothèse est communément appelée une hypothèse de récurrence.

Pour le type inductif `positive`, le principe de récurrence a la forme suivante :

```
positive_ind :
 forall P : positive -> Prop,
```

```
  (forall p, P p -> P (xI p)) ->
  (forall p, P p -> P (x0 p)) ->
  P 1 ->
  forall x, P x
```

Ce principe fait apparaitre trois cas correspondant aux trois formes possibles d'un nombre positif non-nul. Il est bien adapté pour raisonner sur les fonctions qui effectuent des appels récursifs sur les sous-termes directs de l'argument, mais ce n'est pas le cas pour nous, puisque les appels récursifs se font sur les appels de deuxièmes rang.

Un principe de récurrence mieux adapté devrait avoir sept cas, comme ceux qui apparaissent dans la fonction `sqrt`. Ce principe aurait la forme suivante :

```
sqrt_ind :
 forall P : positive -> Prop,
 P 1 -> P 2 -> P 3 ->
 (forall p, P p -> P (x0 (x0 p))) ->
 (forall p, P p -> P (x0 (xI p))) ->
 (forall p, P p -> P (xI (x0 p))) ->
 (forall p, P p -> P (xI (xI p))) ->
 forall x, P x
```

Ce principe se démontre assez aisément dans Coq en utilisant le principe de récurrence `positive_ind` pour la propriété suivante (suivant une technique déjà présentée dans [4]) :

$$\texttt{P x} \wedge \texttt{P (x0 x)} \wedge \texttt{P (xI x)}$$

Voici un script de preuve qui permet cette démonstration.

```
Theorem sqrt_ind :
 forall P : positive -> Prop,
 P 1 -> P 2 -> P 3 ->
 (forall p, P p -> P (x0 (x0 p))) ->
 (forall p, P p -> P (x0 (xI p))) ->
 (forall p, P p -> P (xI (x0 p))) ->
 (forall p, P p -> P (xI (xI p))) ->
 forall x, P x.
intros; assert (P x ∧ P (x0 x) ∧ P (xI x)).
elim x; intuition.
intuition.
Qed.
```

Il est possible de systématiser la construction de principes de récurrences adaptés pour le raisonnement sur les fonctions récursives, comme cela est décrit dans [3].

Le point le plus technique de la démonstration est de démontrer que les calculs effectués à l'intérieur de la fonction `sqrt_aux` sont correct, sous l'hypothèse que les calculs effectués

dans les appels récursifs sont déjà corrects. Ces calculs se font généralement bien, mais il faut faire attention parce que l'existence de carrés de variables dans les formules rend délicate l'utilisation de la procédure principale de raisonnement sur les inégalités, qui ne traite que l'arithmétique de Presburger, où les variables ne peuvent être multipliées que par des constantes entières. L'énoncé de ce lemme a la forme suivante :

```
Lemma sqrt_aux_correct :
  forall x, ge4 x ->
  forall s' r', s'*s'+r' = Zpos (div4 x) ∧
       Zpos (div4 x) < (s'+1)*(s'+1) ->
  forall s r, (s,r) = sqrt_aux s' r' (mod4 x) ->
  s*s+r= Zpos x ∧ Zpos x < (s+1)*(s+1).
```

Le prédicat `ge4` exprime simplement que le nombre `x` est plus grand que 4. Nous utilisons une fonction `div4` pour décrire la division par 4. Cette fonction est totale et retourne une valeur inhabituelle pour les arguments inférieurs à 4, parce qu'il n'y a pas de donnée dans le type `positive` pour représenter le nombre zéro.

Avec le lemme `sqrt_aux_correct` et le principe de récurrence adapté, on arrive aisément au théorème suivant :

```
Theorem sqrt_correct :
  forall x s r, (s,r)=sqrt x ->
    s*s+r=Zpos x ∧ Zpos x < (s+1)*(s+1).
```

On peut éviter d'avoir à construire un principe de récurrence spécialisé en définissant directement une fonction typée à l'aide de types dépendants, dont le type exprime les bonnes spécifications sur les valeurs retournées. Cette fonction pourrait avoir le type suivant :

```
Inductive sqrt_data (n:Z) : Set :=
    c_sqrt : forall s r:Z, n = s * s + r ->
                      0 <= r <= 2 * s -> sqrt_data n.

sqrt_rempos : forall p: positive, sqrt_data (Zpos p).
```

C'est l'approche que nous avons utilisée pour la fonction fournie dans les bibliothèques de Coq. Néanmoins l'approche utilisée ici présente l'avantage de mieux mettre en valeur les parties communes de l'algorithme pour les différents cas.

# 3 racine cubique

Pour calculer la racine cubique, le même procédé s'applique, on détermine encore un chiffre à chaque itération, mais il s'agit maintenant de grouper les chiffres de la donnée par groupes de trois. La formule utilisée pour déterminer le nouveau chiffre est plus complexe, puisqu'elle fait aussi intervenir les carrés de la racine cubique déjà calculée.

On procède donc de la manière suivante à chaque itération : on décompose le nombre $N$ de telle manière que $N = N' \times \beta^3 + N''$ et $N'' < \beta^3$. Si $v'$ et $r'$ sont la racine cubique et le reste pour $N'$, on accole le nombre $N''$ au reste $r'$, puis l'on cherche le plus grand chiffre $c$ tel que

$$3 \times v^2 \times \beta^2 \times c + 3 \times v \times \beta \times c^2 + c^3$$

est plus petit que cette valeur. Lorsque la base est 2, le calcul est légèrement plus simple, puisqu'il s'agit uniquement de comparer la valeur $12 \times v^2 + 6 \times v + 1$ avec $8 \times r' + N''$.

Le nombre de cas à considérer dans la fonction récursive croît encore. Nous avons maintenant 7 cas de base (sans appels récursifs) et 8 cas récursifs. La fonction prend la forme suivante :

```
Definition cubic_aux (v' r':Z)(x:positive) : Z * Z :=
  let n'' := mod8 x in
  if Z_le_gt_dec (12*(v'*v')+6*v'+1)(8*r'+n'') then
    (2*v'+1, (8*r'+n'')-(12*(v'*v')+6*v'+1))
  else
    (2*v', 8*r'+n'').

Fixpoint cubic (x:positive) : Z*Z :=
  match x with
    1%positive => (1,0)
  | 2%positive => (1,1)
    3%positive => (1,2)
  | 4%positive => (1,3)
    5%positive => (1,4)
  | 6%positive => (1,5)
    7%positive => (1,6)
  | xO(xO(xO p)) => let (v,r) := cubic p in cubic_aux v r x
  | xO(xO(xI p)) => let (v,r) := cubic p in cubic_aux v r x
  | xO(xI(xO p)) => let (v,r) := cubic p in cubic_aux v r x
  | xO(xI(xI p)) => let (v,r) := cubic p in cubic_aux v r x
  | xI(xO(xO p)) => let (v,r) := cubic p in cubic_aux v r x
  | xI(xO(xI p)) => let (v,r) := cubic p in cubic_aux v r x
  | xI(xI(xO p)) => let (v,r) := cubic p in cubic_aux v r x
  | xI(xI(xI p)) => let (v,r) := cubic p in cubic_aux v r x
  end.
```

La fonction auxiliaire `cubic_aux` est similaire à la fonction `sqrt_aux` que nous avons décrite à la section précédente, tout en reposant maintenant sur deux fonctions `div8` et `mod8` pour la division par 8 et le reste. Les démonstrations utilisent les mêmes séquences de tactiques, bien que l'une des preuves doive couvrir 7 cas tandis que l'autre doit en couvrir 15. En fait, les cas sont à classer en deux grandes familles : soit il s'agit d'un cas de base et alors la preuve se fait rapidement par calcul et vérification (car les cas de base sont en nombre fini),

soit il s'agit d'un cas récursif et les fonction `div4` et `mod4` d'une part et `div8` et `mod8` d'autre part permettent d'abstraire les détails de chaque cas et de retrouver la structure commune à chacune des preuves.

# 4 racine énième

Même si nous avons réussi à décrire la racine carrée et la racine cubique suivant une même structure, il est évident que cette structure ne peut s'utiliser uniformément pour les racines d'ordre plus élevé. En effet, le nombre de cas qui apparait dans chaque fonction est $2^n - 1$ si l'on veut extraire la racine d'ordre $n$. Cette structure est encore moins adaptée si l'on veut disposer d'une fonction unique capable de traiter les racines d'ordre $n$ pour tout $n$. Nous allons devoir mettre en œuvre une technique plus élaborée de définition de fonctions récursives.

## 4.1 Description informelle de l'algorithme

Si nous voulons calculer récursivement la racine énième d'un nombre $x$ positif pour le rang $n$ nous pouvons procéder de la façon suivante :

1. si $x$ est 0, alors la racine est 0 et le reste est 0,
2. si $x$ est non-nul et inférieur à $2^n$, alors la racine est 1, et le reste est $x - 1$,
3. Si $x$ est supérieur à $2^n$, on divise $x$ par $2^n$ pour obtenir un quotient $x'$ et un reste $x''$,
4. on calcule la racine $v'$ et le reste $r'$ pour $x'$,
5. on calcule les valeur $r'' = r' \times 2^n + x''$ et $d = (2 \times s' + 1)^n - (2 \times s')^n$,
6. on compare la valeur de $r''$ avec $d$ :
   - si $r''$ est plus grand, alors le résultat est $2 \times v' + 1$ et le reste est $r'' - d$,
   - sinon le résultat est $2 \times v'$ et le reste est $r''$.

Par souci d'efficacité, il est souhaitable de calculer une fois pour toute la valeur $2^n$ et le polynôme $P_n = y \mapsto (2 \times y + 1)^n - (2 \times y)^n$. En particulier, si nous simplifions le polynôme, il ne contient pas de terme de degré $n$.

Nous ne pouvons plus utiliser directement la structure du type inductif `positive` pour décrire l'algorithme, parce que le nombre de cas qu'il faut faire apparaitre pour respecter les contraintes de la récurrence structurelle varie avec le degré de la racine $n$. Nous avons préféré décrire notre algorithme directement comme un algorithme travaillant sur les nombres entiers, même s'il ne retourne pas de valeur significative lorsque le premier argument est négatif et lorsque le second argument est négatif ou nul. Notre algorithme est probablement mieux adapté pour travailler sur des nombres dont la représentation interne est la base 2, parce que l'opération de division par $2^n$ y est alors mise en œuvre de façon efficace, mais il s'adapte sans difficulté à d'autres bases.

## 4.2 L'outil `Recursive Definition`

La fonction de racine énième que nous considérons n'est pas une fonction récursive structurelle comme celles que nous avons vues dans les sections précédentes. Dans la description informelle de l'algorithme, nous voyons que l'appel récursif se fait sur la valeur $x'$, le quotient de la division par $2^n$. Cette valeur est obtenu par un appel à une fonction plutôt que par une construction de filtrage et $x'$ n'apparait donc pas clairement que comme un sous-terme de la structure de $x$. Nous dirons que de telles fonctions sont des fonctions *récursives générales.*

Le système Coq fournit quelques outils pour la programmation de fonctions récursives générales, le plus commun est la programmation par récursion *bien fondée.* Le principe est toujours d'assurer qu'aucune fonction récursive n'entrera dans un calcul infini par une succession interminable d'appels récursifs. Au lieu d'utiliser un argument sur la forme des fonctions définies, comme c'est le cas pour la récursion structurelle, on utilise un argument logique basé sur l'utilisations de relations bien fondées.

Les relations bien fondées sont des relations qui ne présentent pas de suite infinie décroissantes. Lorsque l'on définit une fonction récursive, on peut imposer que les arguments des appels récursifs dans la fonction soient des prédécesseurs de l'argument initial pour une relation bien fondée fixée. Comme la relation n'admet pas de suite infinie décroissante, il ne pourra pas y avoir de suite infinie d'appels récursifs et les calculs infinis seront évités.

Plusieurs méthodes de définitions sont fournies pour définir des fonctions récursives structurées. Nous avons utilisé l'outil `Recursive Definition` que nous avons décrit dans [2] et qui est fourni sur le site internet du système Coq parmi les contributions des utilisateurs.

Dans le prototype actuellement fourni, seules les fonctions à un seul argument et présentant un seul appel récursif sont traitées, pour les autres cas la méthode mise en œuvre dans l'outil peut être appliquée manuellement en suivant le schéma décrit dans [2, 4]. Cet outil permet de définir une fonction récursive en donnant son type, la relation bien-fondée qui contrôle l'appel récursif, la preuve que cette relation est bien fondée, le théorème qui montre que l'argument de l'appel récursif est bien un prédécesseur de l'argument initial (dans la suite ce théorème sera appelé *théorème de décroissance*) et l'équation récursive qui caractérise cette fonction. Cet équation récursive sert à décrire le contenu algorithmique de la fonction.

Un avantage de l'outil est que l'équation récursive est très similaire au programme que l'on écrirait dans un langage de programmation fonctionnel usuel en s'imposant seulement la contrainte d'écrire un programme purement fonctionnel.

Nous l'avons dit, le prototype ne considère pour l'instant que les fonctions récursives à un argument. Ceci n'est pas une contrainte importante, puisque plusieurs arguments peuvent être regroupés dans un seul multiplet. Ici, nous voulons éviter que la fonction récursive recalcule le polynome $P_n$ et la valeur $2^n$ à chaque étape et nous voulons donc passer ces valeurs en argument de la fonction. Notre premier choix est donc de prévoir que l'argument de la fonction soit un triplet :

- la première composante est le nombre dont on veut extraire la racine énième,
- la seconde composante est la liste des coefficients du polynôme $P_n$,
- le troisième argument est le nombre $2^n$.

La clef de la programmation bien fondée est de fournir une relation bien fondée. Il existe quelques relations bien fondées de base et plusieurs outils pour en fabriquer de nouveaux. Par exemple, les bibliothèques de Coq fournissent une relation ternaire `Zwf` telle que "`Zwf` $k$ $v_1$ $v_2$" est satisfait si et seulement si $k <= v_2 \wedge v_1 < v_2$. Un théorème est également fourni pour exprimer que pour tout $k$ fixé la relation binaire "`Zwf` $k$" est une relation bien fondée. Parmi les outils fournis pour construire de nouveaux ordres bien fondés, un outil important est le théorème qui exprime que l'image inverse d'une relation bien fondée par n'importe quelle fonction est également bien fondée. Nous utilisons ce théorème pour définir la relation bien fondée qui contrôle notre fonction récursive. En effet, nous ne comparons que la première composante de deux triplets et vérifions que ces composantes sont reliées par la relation "`Zwf 0`". Ainsi, la relation bien fondée que nous choisissons est l'image inverse de la relation "`Zwf 0`" par la fonction de projection vers la première composante des triplets.

L'outil `Recursive Definition` est prévu pour fournir une aide à l'utilisateur en indiquant quel but devra être démontré pour assurer que l'appel récursif respecte la relation bien fondée. Il suffit d'appeler cet outil en fournissant tous les éléments de la définition, mais un mauvais théorème de décroissance (ici nous fournissons le théorème `I` qui une preuve de `True`) :

```
Recursive Definition nroot_aux (Z*(list Z*Z) -> Z*Z)
  (fun (x y:Z*(list Z*Z)) => Zwf 0 (fst x) (fst y))
  (wf_inverse_image (Z*(list Z*Z)) Z (Zwf 0) (@fst Z (list Z*Z))
     (Zwf_well_founded 0))
  I
  (forall t,
    nroot_aux t =
    let (p, t') := t in
    let (l, twopn) := t' in
    (let (p', p'') := Zdiv_eucl p twopn in
     if Z_le_gt_dec p' 0 then
         if Z_eq_dec p 0 then (0, 0) else (1, p - 1)
       else let (v', r') := nroot_aux (p', (l,twopn)) in
            let aux_v := eval_poly v' l in
              if Z_le_gt_dec aux_v (twopn * r' + p'')
                then (2 * v' + 1, (twopn * r' + p'') - aux_v)
                else (2 * v', twopn * r' + p'')))).
```

Le texte commençant par la ligne "`nroot_aux t =`" décrit la fonction récursive que nous voulons définir. Les lecteurs habitués de la programmation récursive dans les langages de programmation fonctionnels reconnaitront que ce texte est similaire à une définition de fonction usuelle dans l'un de ces langages de programmation.

Notre définition utilise quelques fonctions auxiliaires. La fonction `Zdiv_eucl` effectue la division de son premier argument par le deuxième, lorsque ce dernier est un entier strictement positif et retourne une valeur arbitraire sinon. les fonctions `Z_le_gt_dec` et `Z_eq_dec`

effectuent des comparaisons, la première répond une valeur positive si son premier argument est inférieur ou égal au second, la seconde fonction répond une valeur positive si son premier argument est égal au second argument. la fonction `eval_poly` prend en premier argument un nombre entier et en second une liste d'entiers représentant les coefficients d'un polynôme et retourne la valeur de ce polynôme pour ce nombre entier.

L'outil répond en montrant le but de décroissance que l'on doit résoudre et en indiquant que le terme `I` n'a pas le bon énoncé (`I` est une preuve de la proposition `True`).

```
t : (Z * (list Z * Z))%type
hrec :  ...
p : Z
t' : (list Z * Z)%type
teq : t = (p, t')
l : list Z
twopn : Z
teq0 : t' = (l, twopn)
p' : Z
p'' : Z
teq1 : Zdiv_eucl p twopn = (p', p'')
anonymous : p' > 0
teq2 : Z_le_gt_dec p' 0 = right (p' <= 0) anonymous
============================
 Zwf 0 (fst (p', (l, twopn))) (fst (p, (l, twopn)))
```

Ce but est composé de deux parties. La première partie énumère l'ensemble des hypothèses qui peuvent être faites sur les données du problème. Par exemple on peut supposer `p' > 0` (cette hypothèse est nommée `anonymous`) et l'on peut également supposer que `(p',p'')` est le couple du quotient et du reste de la division de `p` par `twopn` (cette hypothèse est nommée `teq1`). Dans ce but, `twopn` est un nom de variable arbitraire et l'on s'aperçoit rapidement que nous ne disposons pas d'assez d'information : si `twopn` est le nombre 1, alors `p'=p` et nous n'arriverons pas à démontrer le résultat attendu.

Il est donc nécessaire de préciser des informations sur les arguments de la fonction pour arriver à la définir. Nous voulons utiliser cette fonction avec une valeur entière `twopn` qui est une puissance non nulle de 2. Nous sommes donc sûrs que cette valeur est strictement supérieure à 1. Nous allons donc construire une nouvelle forme de multiplet, un multiplet qui contient également une preuve que le troisième argument est supérieur à 1. Nous définissons cette forme de multiplet à l'aide d'une définition d'enregistrement.

```
Record nroot_arg_data : Set :=
  nadc {nad_p1:Z; nad_p2:list Z; nad_p3:Z; nad_p4 : 1 < nad_p3}.
```

Nous devons maintenant adapter notre fonction récursive à la nouvelle forme de l'argument et la commande prend la forme suivante :

```
Recursive Definition nroot_aux (nroot_arg_data -> Z*Z)
  (fun (x y:nroot_arg_data) => Zwf 0 (nad_p1 x) (nad_p1 y))
  (wf_inverse_image nroot_arg_data Z (Zwf 0) nad_p1
     (Zwf_well_founded 0))
  I
  (forall t,
    nroot_aux t =
    let (p, l, twopn, twopn_pos) := t in
    (let (p', p'') := Zdiv_eucl p twopn in
     if Z_le_gt_dec p' 0 then
         if Z_eq_dec p 0 then (0, 0) else (1, p - 1)
       else let (v', r') :=
                nroot_aux (nadc p' l twopn twopn_pos) in
            let aux_v := eval_poly v' l in
              if Z_le_gt_dec aux_v (twopn * r' + p'')
                then (2 * v' + 1, (twopn * r' + p'') - aux_v)
                else (2 * v', twopn * r' + p'')))).
```

L'outil `Recursive Definition` répond en indiquant qu'il faut démontrer le but suivant :

```
  t : nroot_arg_data
  hrec : ...
  p : Z
  l : list Z
  twopn : Z
  twopn_pos : 1 < twopn
  teq : t = nadc p l twopn twopn_pos
  p' : Z
  p'' : Z
  teq0 : Zdiv_eucl p twopn = (p', p'')
  anonymous : p' > 0
  teq1 : Z_le_gt_dec p' 0 = right (p' <= 0) anonymous
  ============================
   Zwf 0 (nad_p1 (nadc p' l twopn twopn_pos))
     (nad_p1 (nadc p l twopn twopn_pos))
```

Nous sommes maintenant dans une meilleure situation, car nous disposons de l'hypothèse supplémentaire `twopn_pos` qui exprime la contrainte nécessaire sur `twopn`.

Dans notre développement nous avons démontré le théorème de décroissance suivant :

```
Theorem dec_thm :
  forall (p p' p'' twopn:Z)(l:list Z)(twopn_pos : 1 < twopn),
  Zdiv_eucl p twopn = (p', p'') -> p' > 0 ->
  Zwf 0 (nad_p1 (nadc p' l twopn twopn_pos))
        (nad_p1 (nadc p l twopn twopn_pos)).
```

Toutes les prémisses de ce théorème sont satisfaites dans le but produit par notre outil. Ceci montre que l'équation récursive décrit bien une fonction récursive qui termine toujours.

Lorsque l'on donne le théorème `dec_thm` comme argument à la place de `I`, l'outil fonctionne normalement et produit plusieurs objets, en particulier une fonction `nroot_aux` et un théorème `nroot_aux_equation` dont l'énoncé est exactement l'équation récursive. Ce théorème est un élément clef pour démontrer que la fonction `nroot_aux` satisfait la spécification de calculer la racine énième d'un nombre.

## 4.3 Théorèmes compagnons

Le type de la fonction `nroot_aux` ne permet pas d'exprimer que la valeur retournée est bien la racine énième de l'argument. Pour l'exprimer nous devons fournir un théorème supplémentaire.

Nous utilisons la fonction `Zpower` fournie dans les bibliothèques de Coq avec la notation "`a ^ b`" pour "`Zpower a b`" pour décrire l'exponentiation d'un nombre

Bien sûr, la fonction n'effectue les calculs attendus que si les arguments supplémentaires de la fonction représentent bien le polynôme $P_n$ et la valeur $2^n$. Nous nous munissons d'une fonction `poly1` qui fabrique le bon polynôme :

```
Definition poly1 (p : positive) : list Z :=
   poly_plus
    (poly_pow (1 :: (2 :: nil)) p)
    (poly_mult1 (- 1) (poly_pow (0 :: (2 :: nil)) p)).
```

La correction de cette fonction est exprimée par le théorème suivant :

```
Theorem poly1_correct:
 forall v n,  eval_poly v (poly1 n) =
    (2 * v + 1) ^ Zpos n - (2 * v) ^ Zpos n.
```

Ce théorème est une conséquence directe des propriétés des opérations d'addition et de multiplication pour les polynômes.

Nous démontrons les théorèmes suivants :

```
Theorem nroot_aux_correct_eq:
 forall p n v r,
 0 <= p ->
 nroot_aux (nadc p (poly1 n) (2^Zpos n) (Zpower2_gt1 n)) =
   (v, r) ->
  v^Zpos n + r = p.

Theorem nroot_aux_correct_interval:
 forall p n v r,
 0 <= p ->
 nroot_aux (nadc p (poly1 n) (2^Zpos n) (Zpower2_gt1 n)) =
```

```
   (v, r) ->
  v^Zpos n <= p < (v + 1)^Zpos n.
```

Comme c'est toujours le cas pour les fonctions définies en programmation bien fondée, les propriétés de ces fonctions se démontrent également en utilisant un principe de récurrence bien fondée. La démonstration s'effectue donc en reposant sur le théorème suivant :

```
well_founded_ind (Zwf_well_founded 0):
forall P : Z -> Prop,
       (forall x : Z, (forall y : Z, Zwf 0 y x -> P y) -> P x) ->
       forall a : Z, P a
```

Ce théorème exprime donc que nous ferons notre démonstration en supposant que la propriété cherchée est déjà satisfaite pour les nombres plus petits que le nombre considéré mais positifs ou nuls. Si nous appliquons ce principe de récurrence à l'argument `p` du théorème, cela signifie en particulier que nous supposerons que la propriété est déjà satisfaite pour les appels récursifs.

Il y a quatre cas dans chaque démonstrations, parce que la fonction `nroot_aux` effectue un test pour déterminer si un appel récursif sera nécessaire puis elle effectue un test supplémentaire dans chacun des cas pour déterminer la valeur finale. Il est donc justifié de faire huit théorèmes auxiliaires avec les énoncés suivants :

```
Theorem nroot_0_eq : forall n, 0 ^Zpos n + 0 = 0.

Theorem nroot_lt_twopn_eq : forall p n, 1^Zpos n + (p-1) = p.

Theorem nroot_gt_twopn_fst_case_eq :
 forall p n p' p'' v' r',
   p = 2^Zpos n*p' + p'' ->
   v'^Zpos n + r' = p' ->
   (2*v'+1)^Zpos n +
    (r'*2^Zpos n +p'' -((2*v'+1)^Zpos n -(2*v')^Zpos n)) = p.

Theorem nroot_gt_twopn_snd_case_eq :
 forall p n p' p'' v' r',
   p = p'*2^Zpos n + p'' ->
   v'^Zpos n + r' = p' ->
   (2*v')^Zpos n + r'*2^Zpos n +p'' = p.

Theorem nroot_0_interval :
  forall n, 0^Zpos n <= 0 < (0+1)^Zpos n.

Theorem nroot_lt_twopn_interval :
forall p n, 0 <= p -> 0 <> p -> 2^Zpos n > p ->
```

```
 1^Zpos n <= p < (1+1)^Zpos n.

Theorem nroot_gt_twopn_snd_case_interval :
 forall p n p' p'' v' r',
   p = 2^Zpos n*p' + p'' ->
   0 <= p'' < 2^Zpos n ->
   v'^Zpos n + r' = p' ->
   v'^Zpos n <= p' < (v'+1)^Zpos n ->
   (2*v'+1)^Zpos n - (2*v')^Zpos n <=
       r' * 2 ^ Zpos n + p'' ->
   (2*v)^Zpos n <= p < (2*v'+1)^Zpos n.
```

Dans ces théorèmes, les variables $v'$ et $r'$ tiennent le rôle du résultat des appels récursifs. Il faut noter que les deux derniers énoncés d'intervalles raisonnent sur des données $v'$ et $r'$ qui doivent vérifier l'égalité $v'^n + r' = p'$. Nous utilisons donc le résultat `nroot_aux_correct_eq` pour le résultat `nroot_aux_correct_interval`.

## 4.4 L'outil `Fix`

Les deux principales qualités de l'outil `Recursive Definition` sont de permettre la définition de fonctions récursives générales en prouvant l'équation récursive associée à ces fonctions et de permettre la description de la fonction récursive en minimisant le recours aux types dépendants. Néanmoins, cet outil a le défaut d'être encore à l'état de prototype. La bibliothèque standard de Coq fournit un autre outil pour construire des fonctions récursives générales, sous la forme d'une fonction à type dépendant `Fix` et d'un théorème associé `Fix_eq`.

```
Fix : forall (A : Set) (R : A -> A -> Prop),
      well_founded R ->
      forall P : A -> Set,
      forall F : (forall x : A,
                  forall rec: forall y : A, R y x -> P y, P x),
      forall x : A, P x

Fix_eq
   : forall (A : Set)(R : A -> A -> Prop)(Rwf : well_founded R)
       (P : A -> Set)
       (F : forall x : A, (forall y : A, R y x -> P y) -> P x),
     (forall (x : A) (f g : forall y : A, R y x -> P y),
      (forall (y : A) (p : R y x), f y p = g y p) ->
                                          F x f = F x g) ->
     forall x : A,
     Fix Rwf P F x =
         F x (fun (y : A) (_ : R y x) => Fix Rwf P F y)
```

Dans cette section, nous allons montrer comment utiliser la fonction `Fix` et le théorème associé pour re-définir la fonction `nroot_aux`. Nous verrons que cette approche demande une plus grande maitrise des types dépendants.

Les arguments de la fonction `Fix` sont dans l'ordre, le type de départ de la fonction que l'on veut définir (nommé `A`), une relation binaire sur ce type (nommée `R`), un théorème démontrant que cette relation binaire est bien fondée, une fonction décrivant le type d'arrivée de la fonction (elle est nommée `P`, c'est une fonction car la fonction `Fix` peut être utilisée pour définir des fonctions à types dépendants) et une fonction (nommée `F`) qui décrit l'algorithme utilisé dans la fonction. Le premier argument (nommé `x`) de la fonction `F` est l'argument de la fonction récursive que l'on veut définir, le second argument (nommé `rec`) est la fonction qui est utilisée pour représenter les appels récursif. Cet argument a lui même un type qui force les appels récursifs à n'avoir lieu que sur les prédécesseurs de l'argument initial pour la relation `R`.

Alors que l'outil `Recursive Definition` autorisait l'utilisateur à fournir une équation récursive dans laquelle les arguments de décroissance n'apparaissaient pas explicitement, la fonction `Fix` exige que l'on fournisse une description de l'algorithme dans laquelle l'argument de décroissance est fourni à la fonction `rec` lorsqu'elle est utilisée. Nous pouvons utiliser le même théorème `dec_thm` que dans la section précédente, mais la description de l'algorithme doit être modifiée pour faire apparaitre les éléments nécessaire à l'application de ce théorème.

Le premier élément manquant est une hypothèse de la forme

```
Zdiv_eucl p twopn = (p', p'')
```

En effet notre description initiale de l'algorithme contient le fragment

```
let (p,p') := Zdiv_eucl p twopn in  ...
```

Ceci introduit bien les variables `p` et `p'` dans le programme, mais pas l'égalité nécessaire. Pour obtenir l'égalité nécessaire, nous utilisons une technique déjà décrite dans [4] sous le terme "renforcement minimal de spécification" (en anglais *minimal specification strengthening*) pour construire une fonction qui retourne les même valeurs `p` et `p'` plus une preuve de l'égalité requise :

```
Inductive div_eucl_data (p d:Z) : Set :=
 dedc : forall q r, Zdiv_eucl p d = (q, r) -> div_eucl_data p d.

Definition Zdiv_eucl' (p d:Z) : div_eucl_data p d :=
  (match Zdiv_eucl p d as v
       return Zdiv_eucl p d = v -> div_eucl_data p d with
     (q, r) => fun h => dedc p d q r h
   end (refl_equal (Zdiv_eucl p d))).
```

Le théorème `dec_thm` requiert également une preuve de `p' > 0`, mais cette preuve est fournie naturellement par le test "`Z_le_gt_dec 0 p`" si l'on se trouve dans la branche adéquate, à condition d'utiliser une construction de filtrage au lieu d'une construction `if-then-else`. La fonction que nous allons fournir à `Fix` aura donc la forme suivante :

```
Definition nroot_aux'_F (t : nroot_arg_data) :=
    match t return
      (forall y:nroot_arg_data,
             (Zwf 0 (nad_p1 y) (nad_p1 t))->Z*Z)->Z*Z with
     nadc p l twopn twopn_pos =>
      fun nroot_aux' =>
       let (p', p'', h) := Zdiv_eucl' p twopn in
       match Z_le_gt_dec p' 0  with
         left _ => if Z_eq_dec p 0 then (0, 0) else (1, p - 1)
       | right hgt =>
         let (v', r') :=
              nroot_aux' (nadc p' l twopn twopn_pos)
               (dec_thm p p' p'' twopn l twopn_pos h hgt) in
              let aux_v := eval_poly v' l in
                if Z_le_gt_dec aux_v (r' * twopn + p'')
                  then (2 * v' + 1, (r' * twopn + p'') - aux_v)
                  else (2 * v', r' * twopn + p'')
       end
    end.

Definition nroot_aux' : nroot_arg_data -> Z*Z :=
 Fix (wf_inverse_image nroot_arg_data Z (Zwf 0) nad_p1
           (Zwf_well_founded 0))
   (fun _ => (Z*Z)%type) nroot_aux'_F.
```

Dans cette approche nous avons utilisé à deux reprises des constructions de filtrage dépendant, la première fois dans la fonction `Zdiv_eucl'` et la seconde dans `nroot_aux'_F`. Ces constructions de filtrage dépendant permettent d'identifier les expressions de type `nroot_arg_data` ou `Z*Z` et leur contenu.

La fonction `nroot_aux'` satisfait la même équation récursive que la fonction `nroot_aux`. Nous prouvons cette équation facilement à l'aide du théorème `Fix_eq`. Après réécriture à l'aide de ce théorème, deux parties apparaissent dans la preuve. Dans la première partie, il faut montrer que la fonction `nroot_aux'_F` et l'algorithme décrit dans l'équation récursive coincident. La seule différence entre ces deux parties est l'utilisation de la fonction `Zdiv_eucl'` dans un cas et de la fonction `Zdiv_eucl` dans l'autre. Du point de vue algorithmique, la différence entre ces deux fonctions est anecdotique, et il est rapide de démontrer que les similitudes entre ces deux fonctions suffisent à assurer l'égalité.

La deuxième partie consiste à démontrer l'hypothèse du théorème `Fix_eq`. Cette hypothèse est facile à démontrer car il s'agit simplement de démontrer que la même expression effectue les même calculs de part et d'autres. Cette démonstration ne se fait pas en une seule étape parce que la théorie supportée dans le calcul des constructions n'est pas extensionnelle. Néanmoins, nous avons démontré dans [1] que cette démonstration se faisait systématiquement, au point d'être pratiquement automatisable (nous disposons d'un prototype).

Puisque la fonction `nroot_aux'` satisfait la même équation que `nroot_aux` on pourra démontrer les mêmes propriétés pour cette fonction.

# 5 La fonction principale

Les fonctions `nroot_aux` et `nroot_aux'` ne sont que des fonctions auxiliaires qui travaillent avec un polynôme et un diviseur arbitraires. Il est nécessaire d'appeler l'une de ces fonctions avec les bons arguments pour calculer effectivement une racine énième. En particulier, le théorème `Zpower2_gt1` est une preuve que $2^n$ est toujours strictement supérieur à 1 (quand $n$ est un nombre strictement positif)

```
Definition nroot (p n : Z) :=
   match n with
     Zpos v => nroot_aux
      (nadc p (poly1 v) (2 ^ (Zpos v))  (Zpower2_gt1 v))
    | _ => (0, 0)
   end.
```

La correction de cette fonction est assurée par le théorème suivant qui se démontre aisément à l'aide des théorèmes précédents.

```
Theorem nroot_correct:
 forall p n v r,
 0 <= p ->  0 < n ->
 nroot p n = (v, r) ->
    v ^ n + r = p ∧ ( v ^ n <= p < (v + 1) ^ n ).
```

# 6 Travaux similaires

De nombreux chercheurs travaillent sur la certification de l'arithmétique des ordinateurs dans le monde, en se concentrant sur les opérations flottantes [10, 9]. Une partie de ces recherches est soutenue par les fabricants de microprocesseurs [11, 15]. En particulier, nous avons travaillé sur la certification d'un algorithme de calcul de racines carrées sur les grands nombres qui généralise l'algorithme utilisé dans le présent article pour obtenir une approche de type *diviser pour régner* [5]. La généralisation de l'algorithme utilisé ici pourrait également mener à une implémentation efficace pour les grands nombres.

Une deuxième facette de cet article est la description qui est faite des techniques disponibles pour décrire des algorithmes récursifs généraux. Les deux techniques que nous avons décrites ici ont en commun de reposer sur la notion de relation bien-fondée. D'autres solutions ont été proposées qui reposent sur une utilisation plus forte de types dépendants. Ana Bove défend le point de vue que l'on décrive la terminaison de toute fonction récursive par un prédicat inductif défini pour la circonstance [6]. Cette approche présente l'avantage de

permettre un pouvoir expressif important et en particulier de permettre la description de fonction récursives partielles. Ces travaux ont surtout été décrits dans le cadre de la théorie des types de Martin-Löf mais leur transposition dans le calcul des constructions se fait assez aisément. Par exemple, nous avons utilisé cette technique dans la description d'algorithmes calculant sur les nombres rationnels [14]. McBride et McKinna défendent un point de vue similaire et vont plus loin en proposant une nouvelle construction de filtrage adaptée à la programmation utilisant ces prédicats inductifs [13]. Ces approches sont puissantes et élégantes, mais nous pensons que l'approche fournie avec l'outil `Recursive Definition` apporte une contribution significative en réduisant la compétence requise dans la manipulation de types dépendants, que nous considérons un obstacle à la diffusion des techniques de formalisation logicielle en théorie des types.

D'autres systèmes de démonstration formelle fournissent des moyens de description de fonctions récursives, souvent dans le contexte de types inductifs ou en se reposant sur une théorie de relations nœtheriennes ou bien fondées [17, 16, 12]. Dans ces systèmes, les types dépendants ne sont pas fournis (ou pratiquement pas) et la difficulté d'apprentissage est moindre.

# 7 Conclusion

L'algorithme que nous avons décrit pour la racine énième n'utilise pas du tout la structure binaire utilisée pour la représentation des nombres, il est donc adaptable à n'importe quelle autre base et on pourra réutiliser la même étude pour la vérification formelle d'un algorithme travaillant dans une base arbitraire, en particulier les algorithmes reposant directement sur la représentation usuelle de l'arithmétique des ordinateurs. Néanmoins, il sera préférable de décrire un algorithme qui détermine au moins un chiffre à chaque itération, alors que l'algorithme décrit ici ne détermine qu'un bit à chaque itération.

Au delà des qualités intrinsèques de l'algorithme, cet article est également une présentation didactique de techniques qui permettent la définition de fonctions récursives générales dans le calcul des constructions.

# Références

[1] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics : 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.

[2] Antonia Balaa and Yves Bertot. Fonctions récursives générales par itération en théorie des types. In *Journées Francophones pour les Langages Applicatifs*, January 2002.

[3] Gilles Barthe and Pierre Courtieu. Efficient reasoning about executable specifications in Coq. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Proceedings of TPHOLs'02*, volume 2410 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002.

[4] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development, Coq'art : the calculus of inductive constructions*. Texts in Theoretical Computer Science : an EATCS series. Springer-Verlag, 2004.

[5] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of gmp square root. *Journal of Automated Reasoning*, 22(3–4) :225–252, 2002.

[6] A. Bove. General recursion in type theory. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, International Workshop TYPES 2002, The Netherlands*, number 2646 in Lecture Notes in Computer Science, pages 39–58, March 2003.

[7] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly and associates, 2000.

[8] Coq development team. *The Coq Proof Assistant Reference Manual, version 8.0*, 2004.

[9] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics : 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 169–184. Springer-Verlag, September 2001.

[10] John Harrison. *Theorem Proving with the Real Numbers*. Distinguished dissertations. Springer-Verlag, London, 1998.

[11] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics : 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, 1999. Springer-Verlag.

[12] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-aided reasoning : an approach*. Kluwer Academic Publishing, 2000.

[13] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

[14] Milad Niqui and Yves Bertot. Qarith : Coq formalization of lazy rational arithmetic. In *Types for proofs and programs*, number 3085 in Lecture Notes in Computer Science. Springer-Verlag, 2004.

[15] David M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the AMD K5 Floating-Point Square Root Microcode. *Formal Methods in System Design*, 14(1) :75–125, January 1999.

[16] Natarajan Shankar, Sam Owre, and John M. Rushby. A tutorial on specification and verification using PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1993. (Beta Release).

[17] Konrad Slind. Function definition in higher order logic. In *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*. Sprinter Verlag, August 1996.

[18] Simon Thompson. *Haskell, the craft of functional programming.* Addison-Wesley, 1996.

[19] Pierre Weis and Xavier Leroy. *Le Langage Caml.* Dunod, 1999.

# Table des matières

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399